CS103L SPRING 2020

UNIT 5: ARRAYS

TEXT

LEARNING OBJECTIVES

- Understand definition of data structure
- Understand need for arrays
- Understand how arrays are created
- Understand how arrays are used
- Understand how arrays are stored in memory
- Understand multidimensional arrays
- Understand how simple images can be stored/manipulated as arrays

TEXT

DEFINITION OF DATA STRUCTURE

- Data structure: “In computer science, a data structure is a particular way of organizing data in a computer so that it can be used efficiently.” - Wikipedia
- The nature of some data, and the way we need to accesses it often requires some structure, or organization to make things efficient (or even possible)

TEXT

MOTIVATION FOR ARRAYS

- Create a game with 1 to N players…
- At some point update the scores…
- What is the problem here?

```
…
player1_score += 1;
player2_score += 1;
player3_score += 1;
…
```

TEXT

WITH ARRAYS

- If N is some constant for the maximum # of players
- If you need an individual score outside of the loop:
  - cout << “Player 2: “ << scores[2];

```
int scores[N];
for(i=0;i<N;i++)
{
   scores[i] += 1;
}
```

TEXT

FORMAL DEFINITION

- Arrays are our first data structure that make it easy to access an arbitrary number of like items
- Informal: collection of variables of the same type accessed by an index
- In C/C++ arrays are statically sized, contiguously allocated collections of homogeneous data elements
  - Homogeneous data elements: all elements are the same type (int, char, etc.)
  - Contiguously allocated: one element after another
  - Statically sized: size of the collection must be known at creation and can not change

ARRAY ALLOCATION

- Arrays get a name (like any other variable)
- Individual elements accessed using [] operator and an index

SOME EXAMPLES

- char c[8];
- int N[8];
- double D[8];

Memory Address	Data
0	0x68
1	0x69
2	0x20
3	0x43
4	0x53
5	0x31
6	0x30
7	0x33

c

Memory Address	Data
200	0x68456712
204	0x69789056
208	0x20564523
212	0x43671423
216	0x53898647
220	0x31887567
224	0x30435678
228	0x33875675

N

Memory Address	Data
400	0x6845671214325678
408	0x6978905609567543
416	0x2056452334565430
424	0x4367142356789632
432	0x5389864754673452
440	0x3188756723456223
448	0x3043567887543211
456	0x3387567554346357

D

USING ARRAYS

- Of course we don’t see the memory
- arry[index] can be treated like any other variable
- int scores[3];
- Usually need to initialize:

```
int scores[3];

for(i=0;i<3;i++)

{

   scores[i] = 0;

}
```

Memory Address	Data	“Name"
200	0	score[0]
204	0	score[1]
208	0	score[2]
212	??	??
216	??	??
220	??	??
224	??	??
228	??	??

INITIALIZING ARRAYS

- Several ways to initialize arrays:
  - Programmatically (previous slide)
  - With data:
    - int scores[5] = {0,0,0,0,0};
    - double data[] = {34.54, 123.0, 51.1};

This number (array size) and number of elements in {} must match

If you initialize with {}, you don't need size in between []. It will automatically be set to # of elements

- Can be variables inside {}: int scores[] = {x,y,z};

TEXT

STATIC SIZE ALLOCATION

- Arrays in C++ are always statically sized
- For now, the size must be known at *compile* time

```
double data[10];

#define MAX_PLAYERS 5

int scores[MAX_PLAYERS];

int pieces[MAX_PLAYERS*2];
```

don't do this!

```
int size;

cin >> size;

double data[size];
```

IN CLASS EXERCISES

- pow2
- echo
- arrayprint
- arraybad

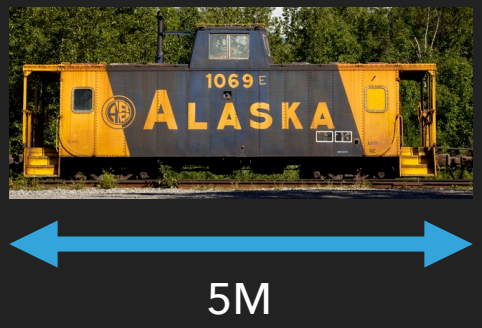
TEXT

ACCESSING ELEMENTS IN AN ARRAY

- C++ has literal/transparent memory model
- Arrays give us our first look at *how* memory is used in a computer
- This will inform your understanding of how many algorithms are implemented

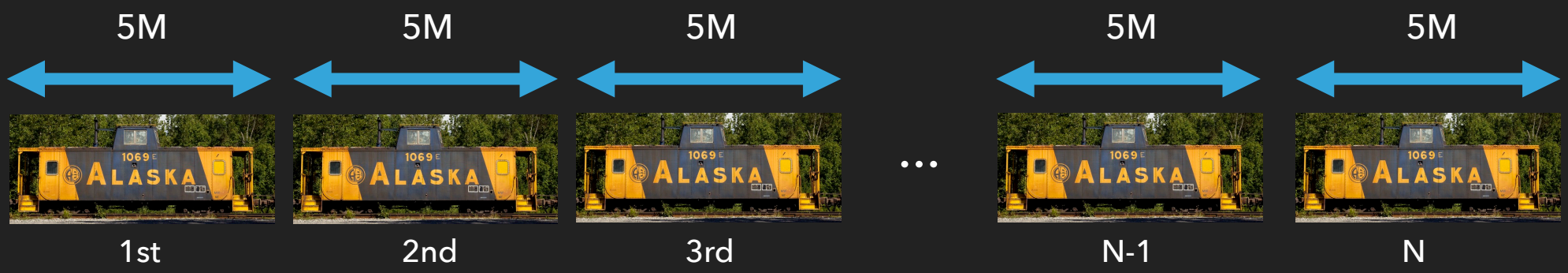
FINDING ELEMENTS

- Think of elements like cars in a train
- #of elements in array = train length
- Element size (e.g int = 4 bytes) = train car length



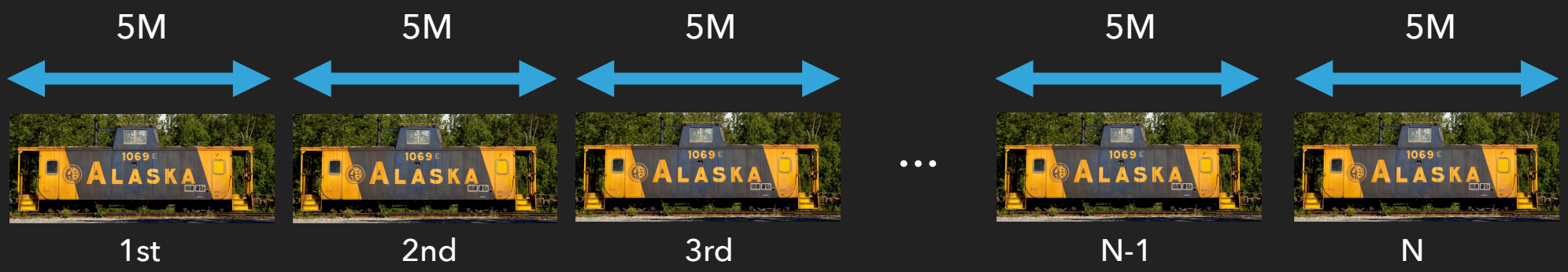
FINDING ELEMENTS

- If each car is 5M long, where does the n^{th} car start?



FINDING ELEMENTS

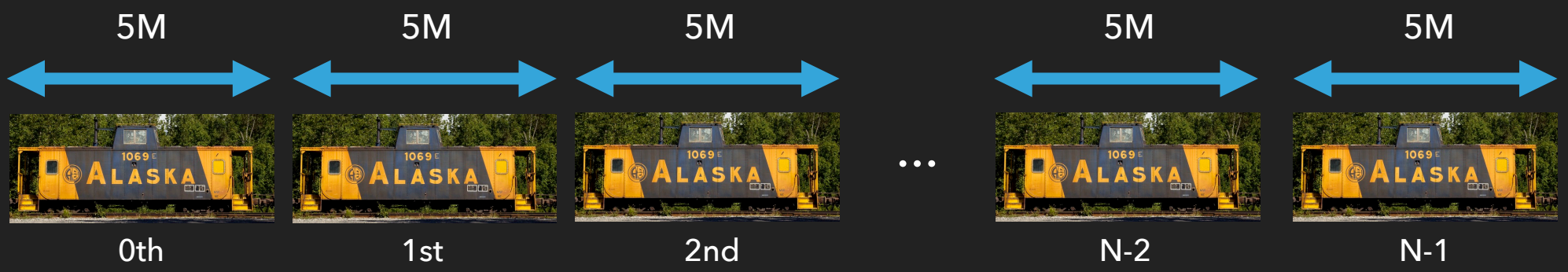
- If each car is 5M long, where does the n^{th} car start?



- n^{th} car starts at (n-1)*5
- Generalize?

FINDING ELEMENTS

- If each car is 5M long, where does the n^{th} car start?



- Start at zero like a good computer
- n^{th} car starts at n*(element size)

TEXT

C/C++ MEMORY AND ARRAYS

- creating an array sets aside memory
- Memory allocated = n * element size
  - int my_array[50] = 50 * 4 = 200 bytes allocated
- C/C++ does not keep track of where each element is
- Rather, calculates *position* when needed

Memory Address	Data
200	0x68456712
204	0x69789056
208	0x20564523
212	0x43671423
216	0x53898647
220	0x31887567
224	0x30435678
228	0x33875675

FINDING AN ELEMENT IN AN ARRAY

- Ex: int arr[8];
  - Sets aside 8*4 = 32 bytes of memory
- arr (the name of the array) resolves to the address of the first element. Ex: 200
- arr[i] *calculates* how to find the i^{th} element:
  - i*sizeof(int) + start
  - arr[0] = 0*4 + 200 = 200
  - arr[3] = 3*4 + 200 = 212

Memory Address	Data
200	0x68456712
204	0x69789056
208	0x20564523
212	0x43671423
216	0x53898647
220	0x31887567
224	0x30435678
228	0x33875675

ARRAY LIMITS

- In C/C++ what happens when:
  - arr[8] = 8*4 + 200 = 232
  - What do we think happens?
- C/C++ *do not* keep track of array bounds for you
- Syntax is valid, what happens? int x = arr[8]; arr[8]++;
  - Crash
  - Weird operation
  - Nothing bad/works great!

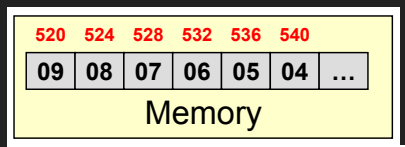
Memory Address	Data
200	0x68456712
204	0x69789056
208	0x20564523
212	0x43671423
216	0x53898647
220	0x31887567
224	0x30435678
228	0x33875675

IN CLASS EXERCISES

- distinct

PASSING ARRAYS AS ARGUMENTS TO FUNCTIONS

- Arrays can be passed to functions
- In function definition use type array_name[] or type *array_name
- void my_function(int* data, int size);
- void function_name(int data[], int size);




```
void add_1_to_array_of_10(int []);
void add_1_to_array(int *, int);
int main(int argc, char *argv[])
{
  int data[10] = {9,8,7,6,5,4,3,2,1,0};
  add_1_to_array_of_10(data);
  cout << "data[0]" << data[0] << endl;
  add_1_to_array(data,10);
  cout << "data[9]" << data[9] << endl;
  return 0;
}
// Example syntax 1
void add_1_to_array_of_10(int my_array[])
{
  int i=0;
  for(i=0; i < 10; i++){
    my_array[i]++;
  }
}
// Example syntax 2
void add_1_to_array(int *my_array, int size)
{
  int i=0;
  for(i=0; i < size; i++){
    my_array[i]++;
  }
}
```

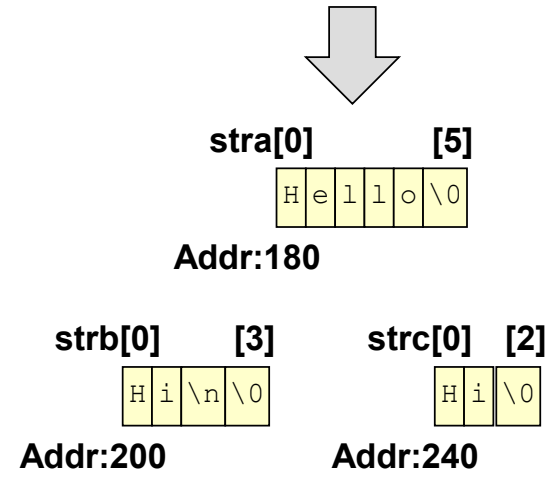
520 520 520 520

C-STRINGS

C-STRINGS

- C needed a way to represent strings: sequences of text characters
- Solution: use an array to hold a sequence of characters (C-string)
- char text[] = “Hello CS103!”;
- C-strings use one byte per character (more-or-less ASCII only)
- What about the length?

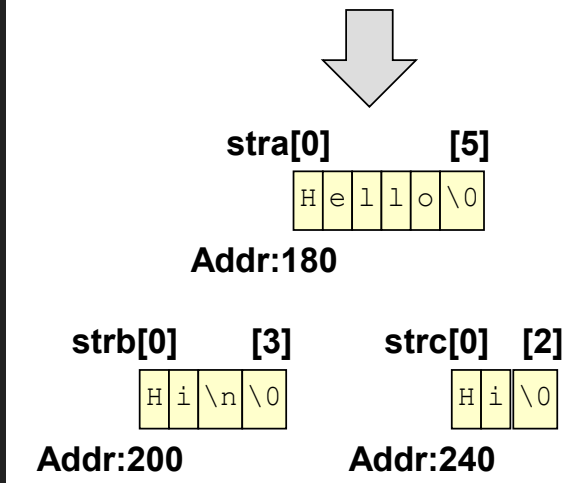
```
#include<iostream>
using namespace std;
int main()
{
  char stra[6] = "Hello";
  char strb[] = "Hi\n";
  char strc[] = {'H','i','\0'};
  cout << stra << strb;
  cout << strc << endl;
  cout << "Now enter a string: ";
  cin >> stra;
  cout << "You typed: " << stra;
  cout << endl;
}
```



C-STRING LENGTHS

- Remember, C/C++ don't keep track of array lengths for you.
- In this code example, how does cout know to only print "Hello"
- C-strings are "delimited", or marked with the NULL character (char value 0) to signify their end.
- Notice we add the \0 to strc in this ex.
- cin will add the \0 for us when reading into C-strings

```
#include<iostream>
using namespace std;
int main()
{
  char stra[6] = "Hello";
  char strb[] = "Hi\n";
  char strc[] = {'H','i','\0'};
  cout << stra << strb;
  cout << strc << endl;
  cout << "Now enter a string: ";
  cin >> stra;
  cout << "You typed: " << stra;
  cout << endl;
}
```



C-STRING HINTS

- When allocating an array for a C-string don't forget space for the NULL character
- Text inside double quotes "like this" makes a string constant including the NULL
- When reading from cin to a C-string buffer, make sure the user doesn't enter too much text as cin will write past the end of your buffer:
  - char buf[25]; cin >> buf; //If user enters <=24 characters we're OK
- If you're creating or modifying a C-string don't forget to include NULL:
  - char buf[5]; buf[0] = 'A'; buf[1] = 'B'; buf[2] = 'C'; buf[3] = 'D'; buf[4] = '\0';

IN CLASS EXERCISES

- strlen
- strcpy
- streq
- CAPITOLIZE

C-STRINGS

- We use C-Strings a lot, so it's important to study how they work
- Lots of connected concepts:
  - representation of text, delimiters
  - arrays
  - pointers (we'll get here)

LOOK UP TABLES

ARRAYS AS LOOK-UP TABLES

- Look up table, or LUT is data structure that performs X → Y mapping function
  - val = Y[X]
- Especially useful for mathematical approximations

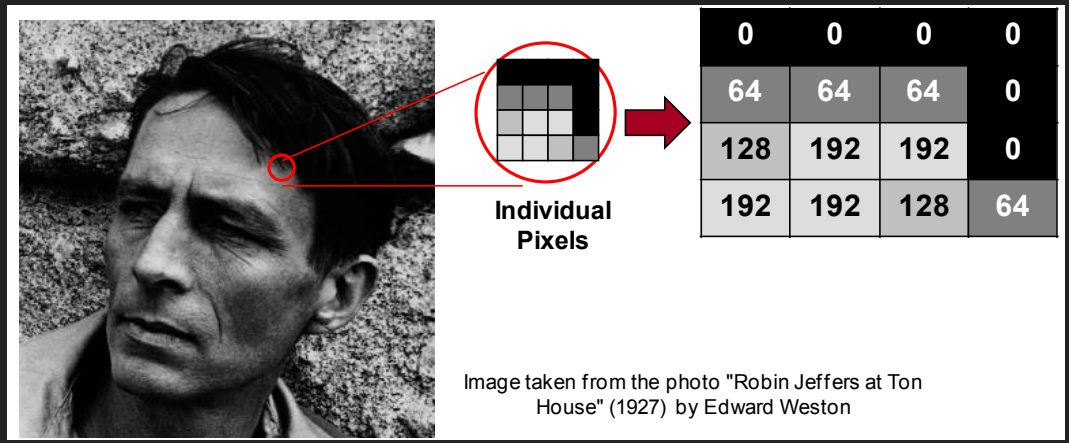
```
// the data
int data[8] = {3, 2, 0, 5, 1, 4, 5, 3};

// The LUT
int squares[6] = {0,1,4,9,16,25};
```

```
// the data
int data[8] = {3, 2, 0, 5, 1, 4, 5, 3};

// The LUT
int squares[6] = {0,1,4,9,16,25};

for(int i=0; i < 8; i++){
  int x = data[i]
  int x_sq = squares[x];
  cout << i << ", " << sq[i] << endl;
}
```

```
// the data
int data[8] = {3, 2, 0, 5, 1, 4, 5, 3};

// The LUT
int squares[6] = {0,1,4,9,16,25};

for(int i=0; i < 8; i++){
  int x_sq = squares[data[i]];
  cout << i << ", " << sq[i] << endl;
}
```

MULTI-DIMENSIONAL ARRAYS

- So far we have used 1-D arrays.
  - Indexed with only one [] operator: int val = array[1];
- Higher dimensional data is natural and common
  - 2D - images, matrix math, linear systems
  - 3D - 3D models, video
  - 4D+ multi-sensor time series



2D ARRAYS

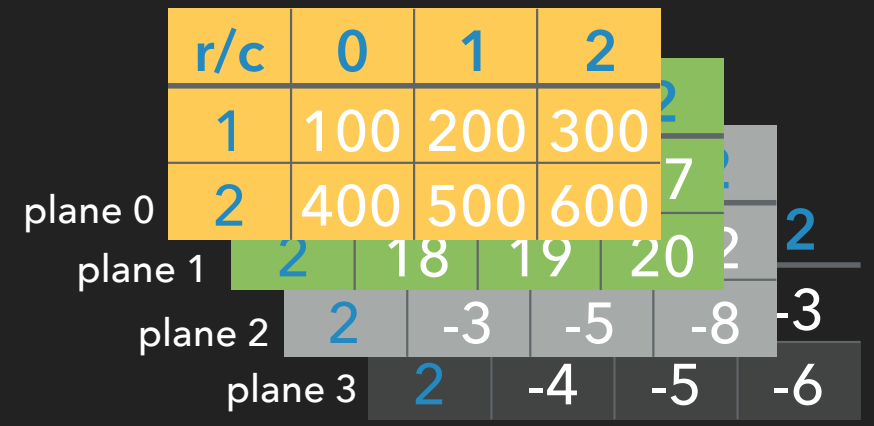
- 2D arrays are index with two [][]
- int 2d[2][3] = {{10,20,30},{40,50,60}}
  - Note: Row/col interpretation is up to *programmer*

r/c	0	1	2
0	10	20	30
1	40	50	60

r/c	0	1
0	10	40
1	20	50
2	30	60

3D ARRAYS

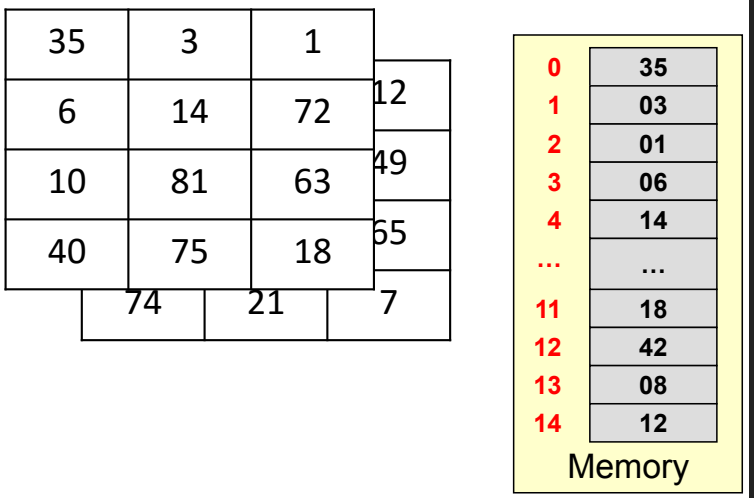
- Declare/use with three []'s
  - unsigned char data[2][3][4]
- [row][column][plane] vs. [plane][row][column] or...
- Interpretation is up to programmer



PASSING MULTIDIMENSION ARRAYS

- Formal parameter: declare name and all dimensions, except the first is optional
- Actual parameter: just use name of the array
- Why… this seems weird…
- Answer:
  - Compiler needs to compute *where* (the address) an element of the array

```
void doit(int my_array[][4][3])
{
  my_array[1][3][2] = 5;
}

int main(int argc, char *argv[])
{
  int data[2][4][3];

  doit(data);
  ...
  return 0;
}
```



TEXT

LINEARIZATION OF ARRAYS

- Arrays are stored in memory
  - “Dimensions” are programmer construct
- Memory is linear, address are not 2D, 3D, etc
- Need to convert array[i][j][k] into a linear address

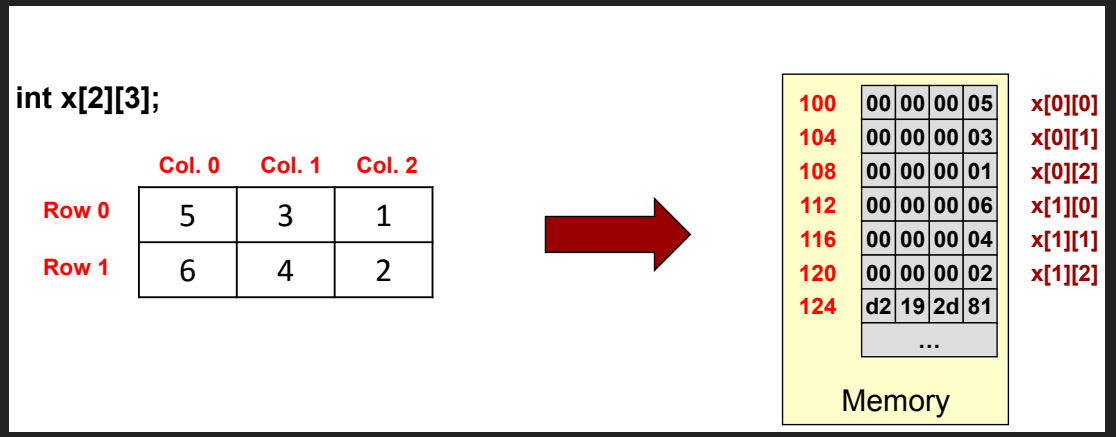
WHAT DO WE MEAN BY LINEARIZE

- Single dimension arrays are already linear

Memory Address	Data
200	0x68456712
204	0x69789056
208	0x20564523
212	0x43671423
216	0x53898647
220	0x31887567
224	0x30435678
228	0x33875675

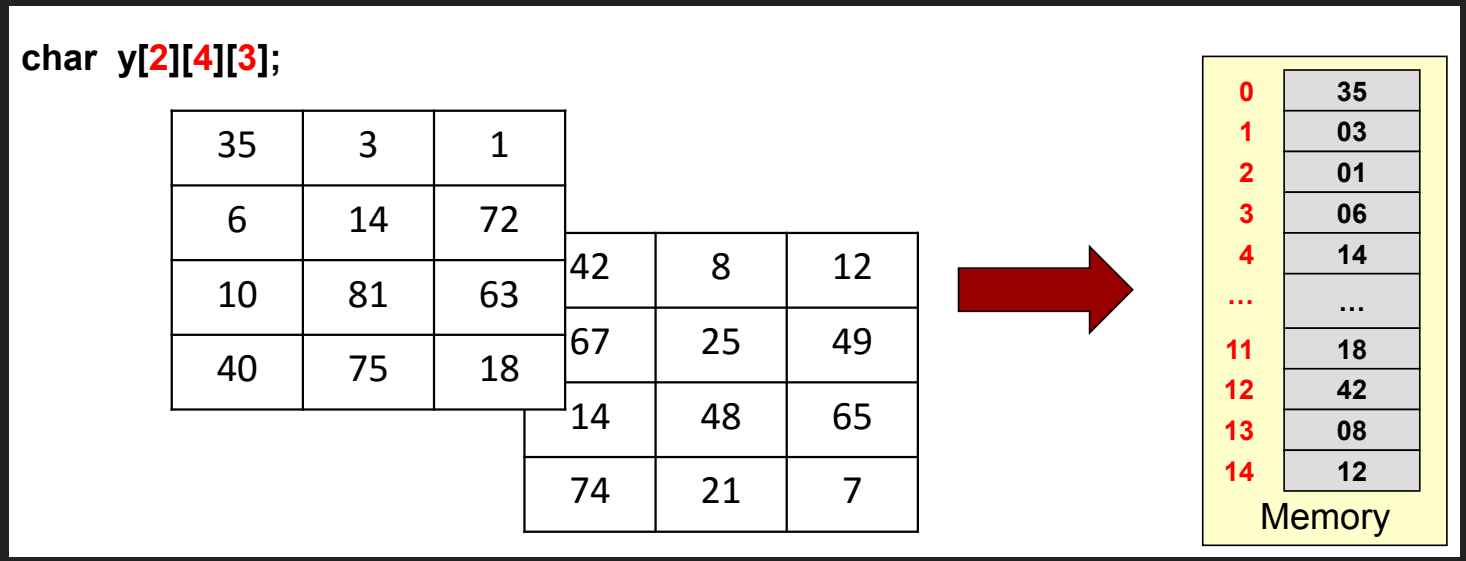
WHAT DO WE MEAN BY LINEARIZE?

- Example: 2D array
- We want to write x[1][2] and get 2
- Linearization is the notion we take [1][2] and figure out where that item would be if we put the items back-to-back
- In C/C++ lower dimensions first to be filled in



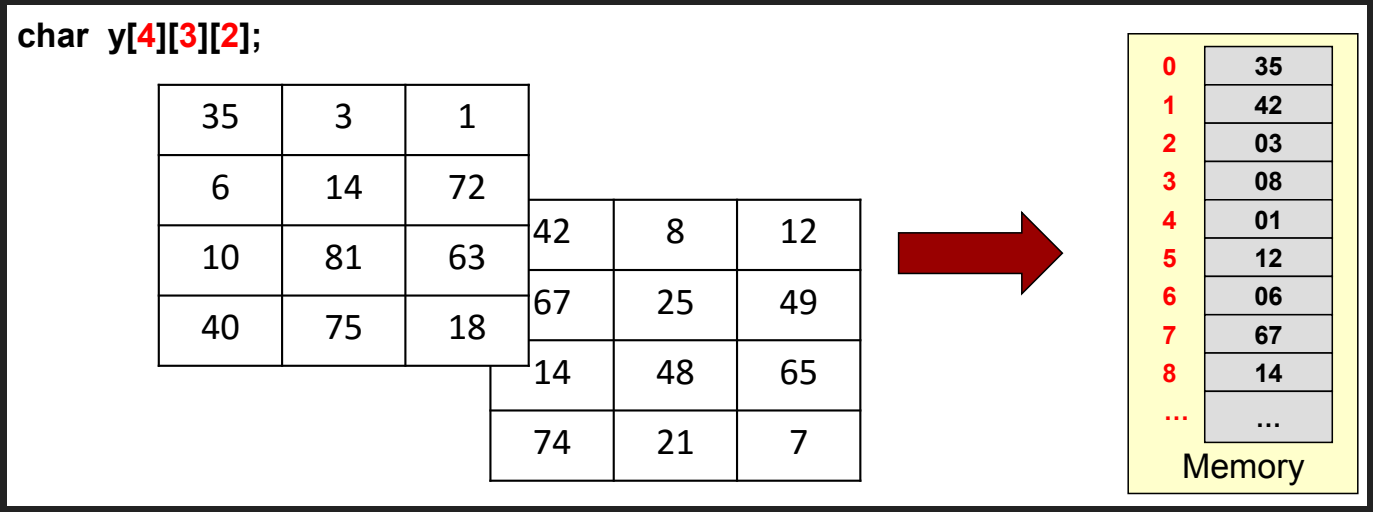
LINEARIZATION OF 3D ARRAY

- All arrays, regardless of dimension are linearized for access
- Ex: 3D



LINEARIZATION ORDER

- Notice, since interpretation is up to us we can change linearization order without changing interpretation

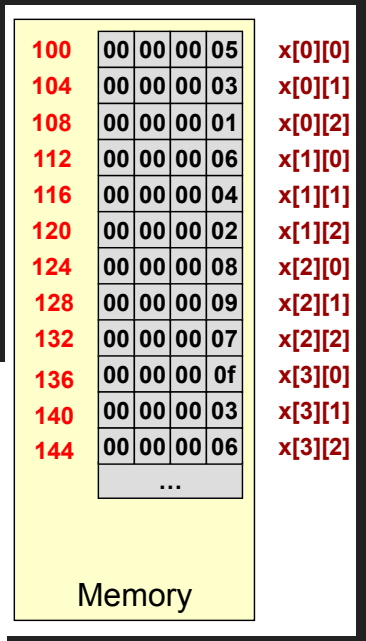


CALCULATING A LINEAR ADDRESS

- Given all of this, how do we calculate a linear address?
- Calculate address of:
- x[i][j]?
- x[1][2]
- 100 + 1*3*4 + 2*4 = 120
- (start address) + ((i*NUMC)+j)*sizeof(int)

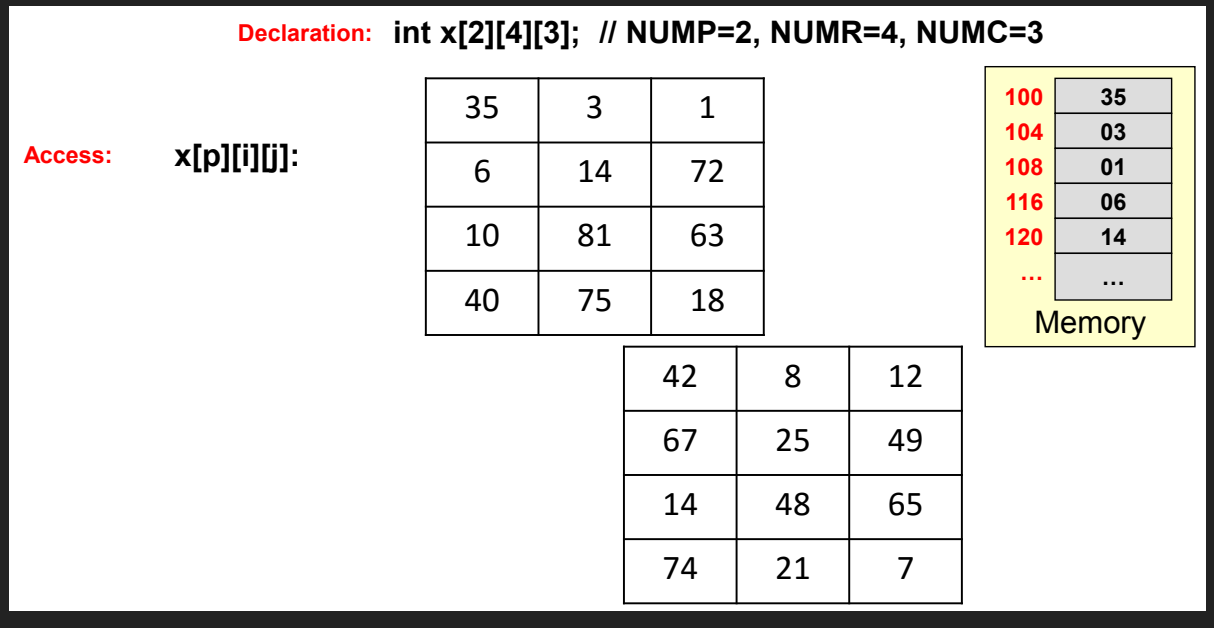
int x[4][3]

	Col. 0	Col. 1	Col. 2
Row 0	5	3	1
Row 1	6	4	2
Row 2	8	9	7
Row 3	15	3	6



CALCULATING LINEAR ADDRESSES

- Formula extends to higher dimensions
- Address = start + (p*NUMR*NUMC+i*NUMC+j)*sizeof(type)



PASSING ARRAYS TO FUNCTIONS

- What does this have to do with passing array to functions?
- Look at previous example? What doesn't appear in formula?
- C/C++ only need lower dimensions to calculate linear address, so highest dimension is optional!

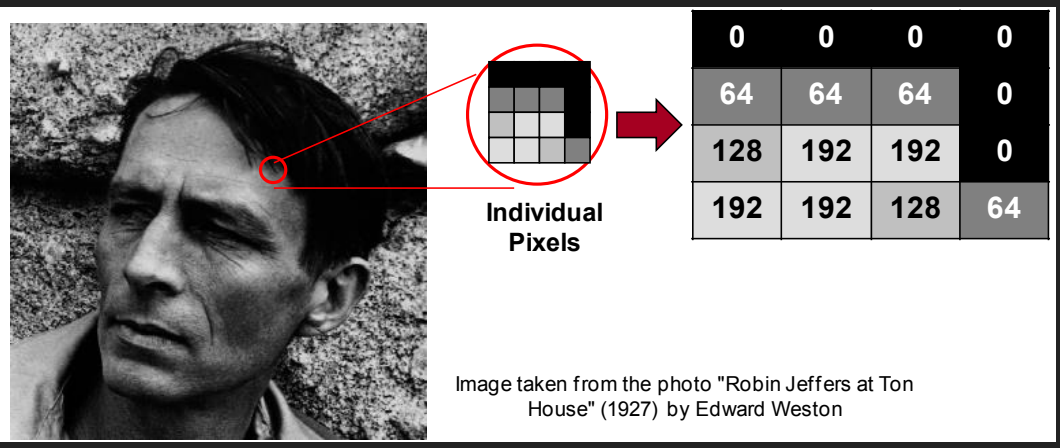
```
void doit(int my_array[][4][3])
{
  my_array[1][3][2] = 5;
}

int main(int argc, char *argv[])
{
  int data[2][4][3];

  doit(data);
  ...
  return 0;
}
```

IMAGES AND BMPLIB

TEXT

MULTIDIMENSIONAL ARRAYS AND IMAGES

- Images in a computer are often represented by 2D or 3D arrays.
- 2D array = B/W image
- 3D array = color images



TEXT

B/W IMAGES AND BMPLIB

- For us B/W images are 256x256 images with pixel values from 0 - 255
- unsigned char image[256][256]
- For reading and display we need a file format
  - We use .bmp
- There is a simple .cpp file that gives us the functions to read/write/display .bmp
- How to use it?

TEXT

MULTIFILE COMPILATION

- BMPLib comes as bmplib.cpp and bmplib.h
- bmplib.h defines prototypes for what functions are available
- bmplib.cpp has the implementation of those functions
- To use put #include “bmplib.h” at the top of your .cpp file
- Then $compile bmplib.cpp my_code.cpp -o my_code

TEXT

DRAWING INTO A BMP

- Once you have this working…
- Declare: unsigned char image[256][256]
- Set values to whatever you’d like
- Write to file

TEXT

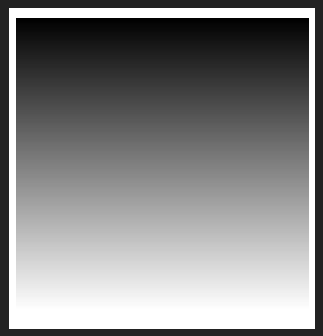
LEARN BY EXAMPLE

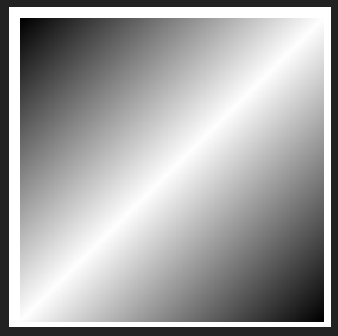
- Download BMPLib and example files to your computer:
- wget http://bits.usc.edu/files/cs103/demo-bmplib.tar
- tar xvf demo-bmplib.tar
- cd demo-bmplib
- make demo

TEXT

DRAWING IN 2D PRACTICE

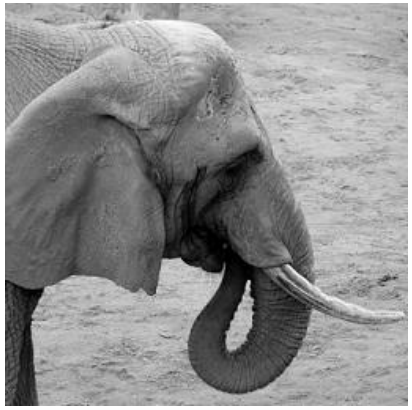
- Modify gradient.cpp to draw a black cross on white background
- Modify gradient.cpp to draw a black X on white background
- Modify gradient.cpp to draw a gradient down the rows
- Modify gradient.cpp to draw a diagonal gradient with black in upper left, white down the diagonal and back to black in lower left





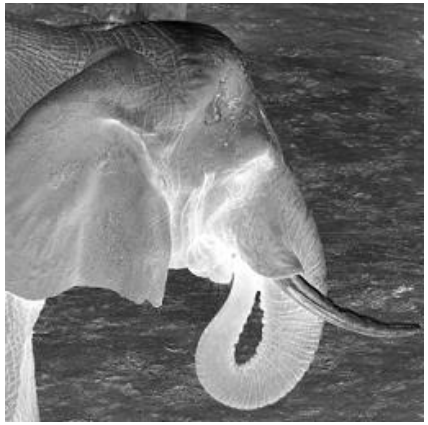
BASIC IMAGE PROCESSING

- We have elephant.bmp in our demo directory
- Example: produce “negative” of Elephant



Original

```
#include "bmplib.h"
int main() {
   unsigned char image[SIZE][SIZE];
   readGSBMP("elephant.bmp", image);
   for (int i=0; i<SIZE; i++) {
      for (int j=0; j<SIZE; j++) {
         image[i][j] = 255-image[i][j];
         // invert color
      }
   }
   showGSBMP(image);
}
```

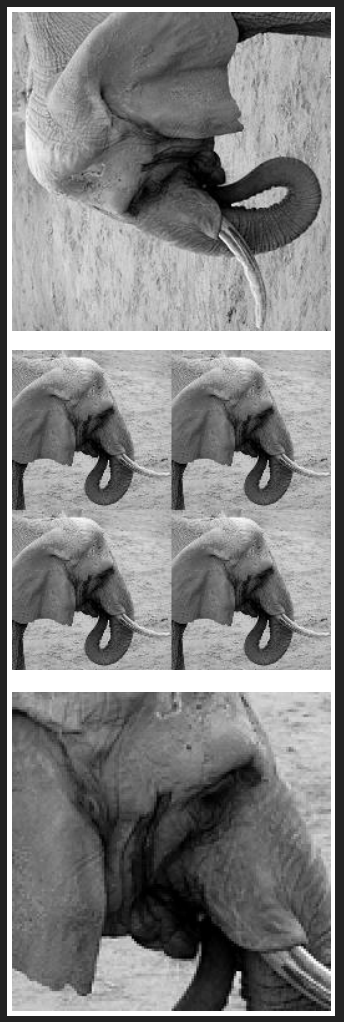


Inverted

TEXT

BASIC IMAGE PROCESSING

- Diagonal flip?
- Tile?
- Zoom?



COLOR IMAGES

- Color Images represented by 3D array: [X][Y][P]
  - Here P = 3 for Red, Green, Blue (RGB)
- So we declare for BMPLIB: unsigned char image[256][256][3]
  - 0 = RED, 1 = GREEN, 2 = BLUE
- Examples:
  - Top: original image
  - Middle: inverted image
  - B/W according to NTSC formula: .299*R + .587*G + .114*B

TEXT

MORE COLOR EXAMPLES

- Glass filter
  - Each destination pixel is from random nearby source pixel
  - http://bits.usc.edu/files/cs103/graphics/glass.cpp
- Edge Detection
  - Gradient/difference filter: each output pixel is abs() of south-west neighbor
  - PA3
- Smooth
  - Each output pixel is average of 8 neighbors
  - PA3
  - http://bits.usc.edu/files/cs103/graphics/smooth.cpp

Original

Smoothed

TEXT

ACKNOWLEDGEMENTS

- Train image from Wikimedia Commons
- Other graphics and examples from Mark Redekopp